Concurrent Processing Memory

Chengpu Wang Zhen Wang Independent researchers

631-974-1078 Chengpu@gmail.com

Abstract

A novel memory with limited processing power and internal connectivity at each element is proposed. This memory carries out parallel processing within itself. Many common algorithms using this memory are discussed. For an array of N items, it reduces the total instruction cycle count of universal operations such as insertion and match finding to ~ 1, and local operations such as filtering and pattern recognition to ~ local operation size. It also reduces the global operations sum and sorting to ~√N and less than ~N instruction cycles respectively. Particularly, it eliminates most streaming activities for data processing purpose on the data bus. Yet it remains general-purposed, easy to use, pin compatible with conventional memory, and practical for implementation.

Keyword: SIMD processors; Parallel Processors; Memory Structures; Performance evaluation of algorithms and systems;

1. Introduction

In database processing, image processing, data stream processing, or modeling, there are a lot simple parallel operations [1][2]. Yet in the current most common CPU/memory bus-sharing architectures [3][4], they are achieved by serial algorithms, in which data are shuffled frequently between the memory unit and the processing unit for the data processing purpose, to contribute to the bus bottle-neck problem [2]. Many solution based on data parallelism has been attempted on this problem. The increase of bus width or caching ability only levitates the problem to certain degrees [2][3][4]. While the connection and simultaneous use of many small functional units [2][5][6][7][8] and grid computing [9] are very impressive in many applications, the mapping of a particular application to the network remains a general problem as MIMD approaches. So far, massive SIMD approaches [2][10][11][12][13][14] are mainly for special applications and they are generally limited by the algorithms to be performed and the ways the elements to be activated,

e.g., even the content address memory [14] achieves a very high degree of parallelism in search, the activation of all memory elements is still done serially in the configuration process, which limits its general usage, especially in a multi-task environment. As a result, the bus-sharing architectures still hold major advantages: (1) They fit well our Human logic, which is based on induction and deduction, both of which are serial in nature. Generally, we only deal with parallel problems as one of the steps of our serial problems. (2) They can have powerful processing units. On the other hand, each processing element in any massive parallel scheme can only have limited capability to make economical sense.

Thus, a solution to the parallel problem is to provide smart memories (A) which carry out parallel operations at where the data are stored, (B) which are still under the control of the processing unit of the bus-sharing architectures, and (C) whose element activation scheme is generic enough for good programmability. The design of a smart memory—the Concurrent Processing Memory, or simply CP memory—is a theoretical attempt to address all these requirements and to provide a unique scheme for (C). Chapter 2 presents the general operational rules and system architecture for the CP memory. The complexity of CP memory elements ranges from simplest and memory cell like as in content movable memory of Chapter 3, to the most complex and CPU like as in math memory of Chapter 4. The former can be viewed as an improved memory, while the latter is an attempt to make the massive SIMD architecture applicable to most massive parallel problems. Other intermediate CPM constructs are shown in Appendix 1. A short discussion is provided in Chapter 4.

2. The Concurrent Processing Memory

2.1 Architecture

In a conventional random access memory [3][4], whose architecture is highly simplified conceptually as Figure 1, each memory element is a register that is connected with a data bus. At each time, only one element whose address is input to an address decoder is activated for either reading from or writing to the data bus. And there is no link between memory elements. A conventional memory is only good at memorizing elements one at a time thus only suitable for serial access.

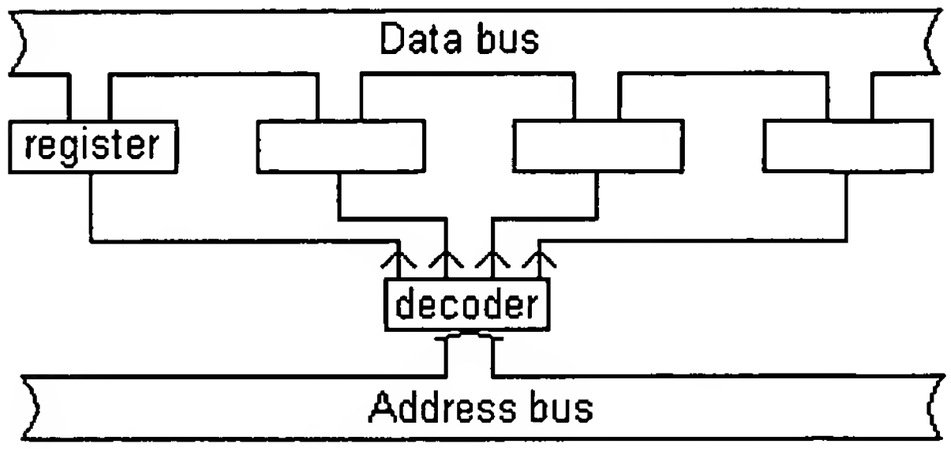

Figure 1: Conventional Memory Architecture

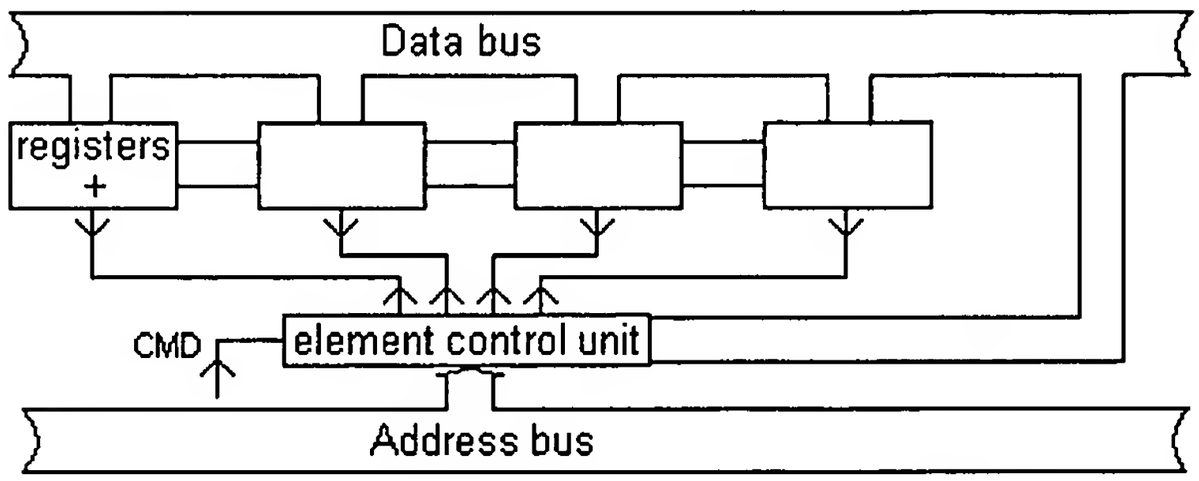

Figure 2: Concurrent Processing Memory Architecture

The basic rules for the proposed CP memory [15], as shown in Figure 2, are:

Rule 1. A CP memory is made of identical elements, each of which has a unique address.

Rule 2. Each element is connected with a data bus.

Rule 3. One element can read from or write to the data bus exclusively.

Rule 4. Multiple elements can be activated concurrently by an element control unit if each of their element addresses is: (1) no less than a start address, (2) no more than an end address, and (3) an integer increment starting from the start address.

Rule 5. Multiple activated elements can read from the data bus concurrently.

Rule 6. Multiple activated elements can be required to identify themselves concurrently. Each element then positively asserts a line which connects the element back to the element control unit.

Rule 7. Each element contains a fixed number of registers.

Rule 8. The neighboring elements are connected so that an element can read at least one register of each of its neighbors.

Rule 9. There is an extra external command pin to indicate that the address and data bus contains whether (1) address and data or (2) an instruction for the memory when it is enabled.

Rule 1, Rule 2, and Rule 3 specify the functional backward compatibility with a conventional random access memory. Rule 4, Rule 5, and Rule 6 define concurrency. Rule 7 and Rule 8 define connectivity. Rule 9 defines processing methodology.

Because the data for parallel problem is usually in the format of array, Rule 4 allows the storing of each array item by multiple neighboring elements, and the concurrent and instant operation upon all array items, or their substructures, or a super-set of the array items which forms a periodical pattern. In fact, Rule 4 distinguishes the CP memory from all other massive SIMD architectures.

A CP memory can be pin-compatible and function-compatible with a conventional random access memory. When it is enabled, an address bit, such as the least significant address bit which is not used by the CP memory, can be dedicated as the command pin of Rule 9. When this pin is negatively asserted, the CP memory behaves exactly like a conventional random access memory; otherwise, the content of the address and data bus is treated as instructions. To a user of the CP memory, sending instructions and getting results from the CP memory is just like writing to or reading from special addresses inside the CP memory.

2.3 System Architecture

The system architecture of a CP memory is shown in Figure 3. As a SIMD structure, all elements receive a same instruction from an I/O control unit through a concurrent bus. Meanwhile, individual addressable registers inside the elements are exclusively accessible through an exclusive bus. The concurrent and the exclusive operations are independent of each other, so that while some registers of one task is being operated on, other registers in the same elements can be simultaneously prepared for other tasks.

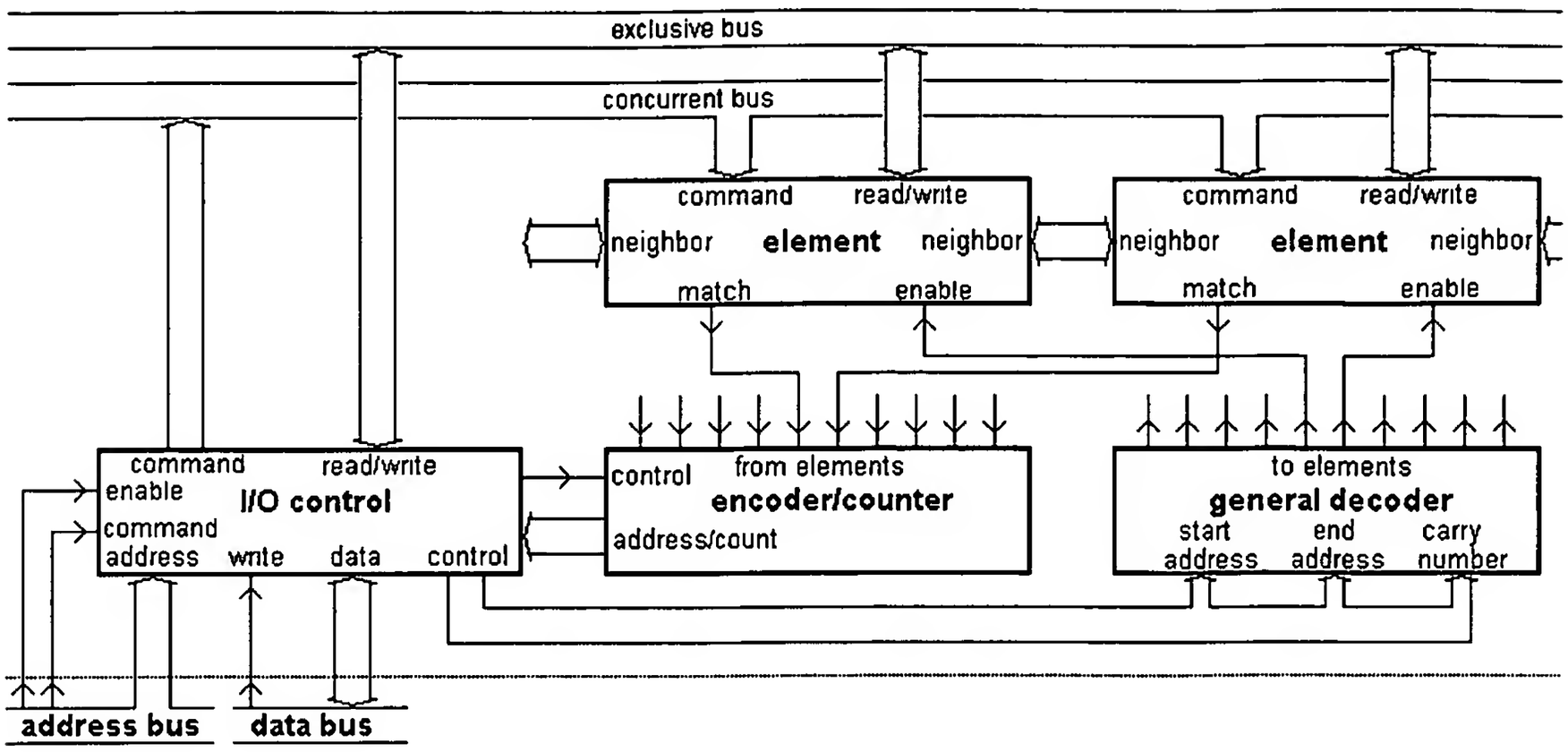

Figure 3: CP Memory System Architecture

Each element may have two bit connections with the element control unit: (1) an enable bit input to the element, which is used for activating the element, and (2) a match bit output from the element, which is used for identifying the element. The element control unit contains: (1) a general decoder to activate the elements; (2) a priority encoder to identify either the highest or the lowest address of the activated element whose match bit output has been positively asserted; and (3) a parallel counter to count the activated element whose match bit output has been positively asserted. The element control unit is also controlled by the I/O control unit.

CP memories also differ in the complexity of system construct. The content movable memory has a constant carry number of 1 for Rule 4, and it has no match bit output and the associated system components.

2.4 General Decoder

The ability to instantly activate elements according to Rule 4 is crucial for the CP memory, which is provided by a general decoder comprising (1) a carry-pattern generator, (2) a parallel shifter, (3) an all-line decoder, and (4) an AND gate array that combines the corresponding bit outputs from the parallel shifter and the all-line decoder.

A carry-pattern generator inputs the carry number to the general decoder and activates all of its bit outputs whose address corresponds to the increments of the carry number. For an

example, a 3/8 carry-pattern inputs binary carry number (C[2] ... C[0]), and positively asserts bit outputs (D[7] ... D[0]) in the following manner:

```
D[0] = 1;

D[1] = !C[2] !C[1]  C[0];

D[2] = !C[2]  C[1] !C[0] + D[1];

D[3] = !C[2]  C[1]  C[0] + D[1];

D[4] =  C[2] !C[1] !C[0] + D[2] + D[1];

D[5] =  C[2] !C[1]  C[0] + D[1];

D[6] =  C[2]  C[1] !C[0] + D[3] + D[2] + D[1];

D[7] =  C[2]  C[1]  C[0] + D[1];
```

The above expression can be generalized for arbitrary number of inputs, and transformed into standard product-of-sum format using either K-map or Quine-McCluskey method [3], and the carry pattern generator can be constructed using corresponding two-level gates. The product-of-sum construct is chosen for expansibility, so that the addition of C[N] input bit appended !C[N] product term to all the existing expressions of (C[N-1] ... C[0]).

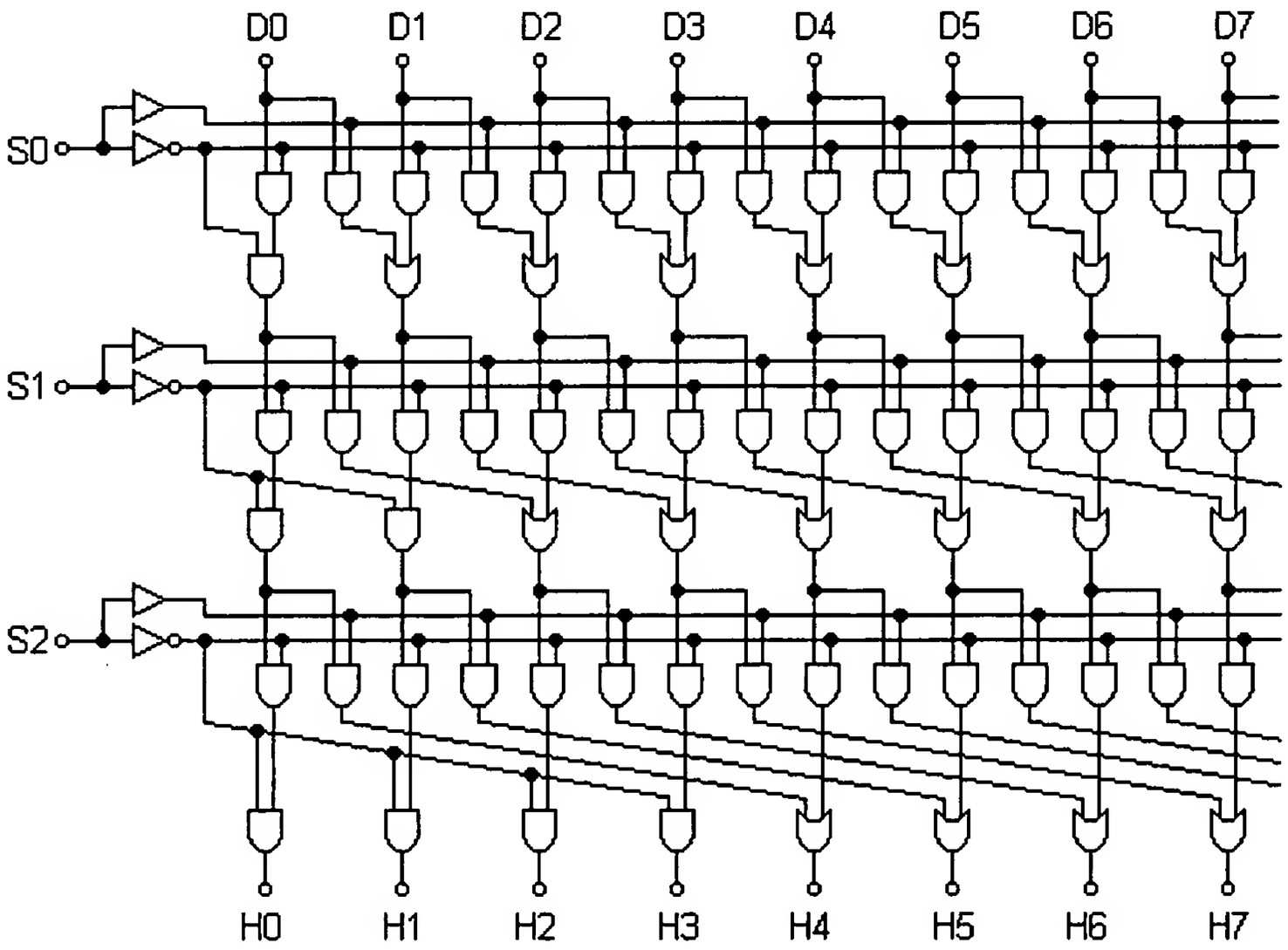

Figure 4: 3/8 Parallel Shifter

The outputs of the carry-pattern generator are shifted toward higher address by the amount of the start address to the general decoder, through a parallel shifter, which inputs a shift amount (S[M] ... S[0]), bit inputs (D[N] ... D[0]), and output bit inputs (H[N] ... H[0]) according to the following equation:

```
IF A => S:   H[A] = D[A - S];

ELSE:        H[A] = 0;
```

Since shifting is accumulative, each S[j] bit input just shifts the bit inputs by the amount of 2^j toward higher input number. For an example, a 3/8 parallel shifter is shown in Figure 4, which inputs a shift amount (S[2] S[1] S[0]), bit inputs (D[7] D[6] D[5] D[4] D[3] D[2] D[1] D[0]), and output bit inputs (H[7] H[6] H[5] H[4] H[3] H[2] H[1] H[0]).

An all-line decoder activates all its bit outputs whose address is less than or equal to its input address, which is the end address to the general decoder in this case. Assuming the bit output is F[E, N], in which E = (E[N-1] ... E[0]) denotes the address of the bit output and N denotes the bit width of the address, an all-line-decoder with address bit width of (N+1) can be built from an all-line-decoder with address bit width of N using the following logic expression of F[E, N]:

```
F[0, 1] = 1;

F[1, 1] = E[0];

F[(0 E[N-1] ... E[0]), N+1] = F[(E[N-1] ... E[0]), N] + E[N];

F[(1 E[N-1] ... E[0]), N+1] = F[(E[N-1] ... E[0]), N]   E[N];
```

The circuit diagram of the 3/8 all-line decoder is shown in Figure 5:

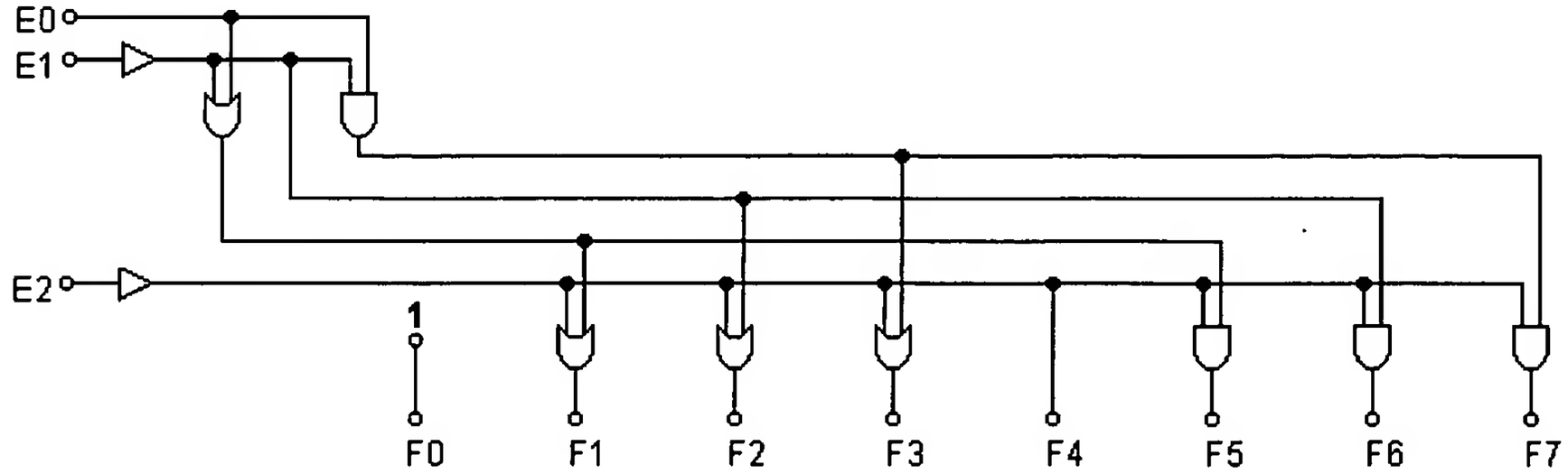

Figure 5: 3/8 All-line Decoder

The bit outputs of the parallel shifter are filtered through AND gates by the corresponding bit outputs of the all-line decoder, before becoming the bit outputs of the general decoder, as shown in Figure 6:

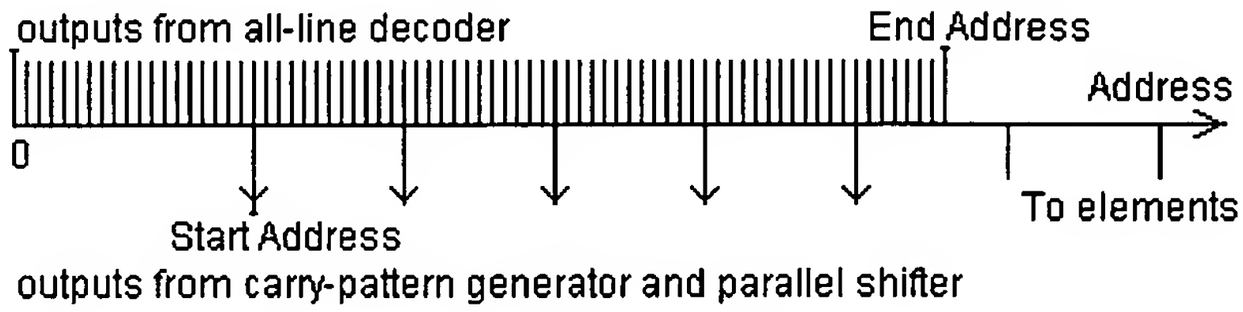

Figure 6: Element Activation Logic

If the carry number is a constant of 1, the start address is input into a first all-line decoder whose outputs are negatively assertive, and the end address is input into a second all-line decoder whose outputs are positively assertive. The corresponding outputs from the two all-line decoders are AND-combined, before becoming the bit outputs of the general decoder.

3 Content Movable Memory

3.1 Construct

A content movable memory is a 1-D CP memory specially designed for inserting, deleting, enlarging, shrinking, or moving data objects within itself, to levitate CPU from most memory management tasks [3]. The structure of a content movable element is shown in Figure 7.

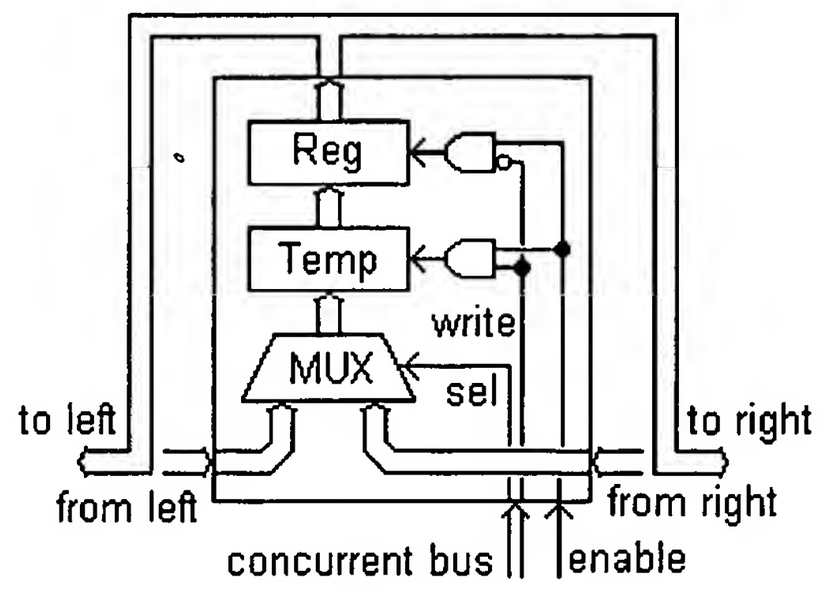

Figure 7: Structure of a Content Movable Element.

Each element has only one addressable register, which can be read by its left and right neighboring memory elements. Each element has an additional temporary register, whose

content can be copied to the addressable register. A multiplexer selects the addressable register of either the left or the right neighbor, to copy to the temporary register. The concurrent bus has only two bits, one to select the output of the multiplexer, and the other to select which registers to be copied. The content of a neighboring addressable register is first copied to the temporary register, then to the addressable register of the element. Thus, the temporary register only needs to remember its content for one clock cycle. The copy operation is enabled by the enable bit input from the element control unit. With the carry number to be a constant of 1 for Rule 4, the content of all the addressable registers within an address range can be moved concurrently in one of the two directions.

3.2 Usage

The content movable memory can be used to manage data objects within itself, such as inserting, deleting, shrinking, enlarging, and moving data objects without overhead such as extensive copying and side-effect such as memory fragmentation. It may allow reference to data objects using abstract IDs instead of the actual addresses which are used to store the objects. It may use containment relationship between the data objects so that: (1) when the size of a contained data object is changed, the container data object is changed accordingly, and (2) when the container data object is moved or removed, all the contained data objects are moved or removed accordingly. In short, it can be a self-managed object memory.

Traditional computer languages require explicitly or implicitly defining the size of a variable before using it, so that a complier can allocate the space for the variable in a conventional memory [3][4]. The improper size of variable is a major source of errors, such as value overflow, value underflow, and unacceptable propagation of precision error, which usually shows up as the mysterious “numerical instability”. Usually allocation in compile-time is far larger than the need, representing a large and frequent waste in memory usage. However, since variable usages at run-time are not predictable, such waste does not generally transfer to immunity of allocation-related problems. The necessity of variable size definition also mixes the implementation methodology with the algorithm goal. When using a content movable memory for a program, the space allocated for a variable can grow and shrink easily according to the need, which brings about the following advantages: (1) a variable will never go out of size, (2) an array is always

dynamic, (3) the memory is always used in the most efficient manner, and (4) the memory is never fragmented, especially for those programming languages that allocate all variables dynamically on heap [16], such as LISP.

3.3 Implementation

At this moment, the content movable memory has the best chance to be realized. Both the temporary and the addressable registers in each memory element of a content movable memory can be made of dynamic memory cells. With 3 gates for each bit, and 2-gate overhead for each memory element, the silicon area per memory element for a content movable memory is very comparable to that of a static random access memory element. By performing a consecutive up/down content move of all used memory elements, the contents of their addressable registers are refreshed locally, concurrently, and instantly. Thus, if technology permits, a content movable memory made of dynamic memory cells can have performance approaching a static random access memory. In addition, the temporary and the addressable registers in each memory element should always have the same content. Thus, a content movable memory may have better error detection & correction capabilities than a normal random access memory.

4. Math Memory and Database Memory

4.1 Element Construct

The element construct of math 1D memory is shown in Figure 8. It has: (1) a status register; (2) an operation register, as the 0th register, (3) an neighboring register, as the 1st register, (4) some data register, as the 2nd, 3rd, etc, registers, (5) an input from the concurrent bus, and (6) neighborhood connections from the neighboring registers of neighboring elements, which has immediately lower or higher addresses. A math 2D memory element has two more neighborhood connections than the math 1D memory element. A database memory element has neither adder, nor carry bit in its status register, nor operation multiplexer; and its “read” output of register multiplexer is connected directly to the “write” input of the bit demultiplexer. Otherwise, these three kinds of memory elements are identical.

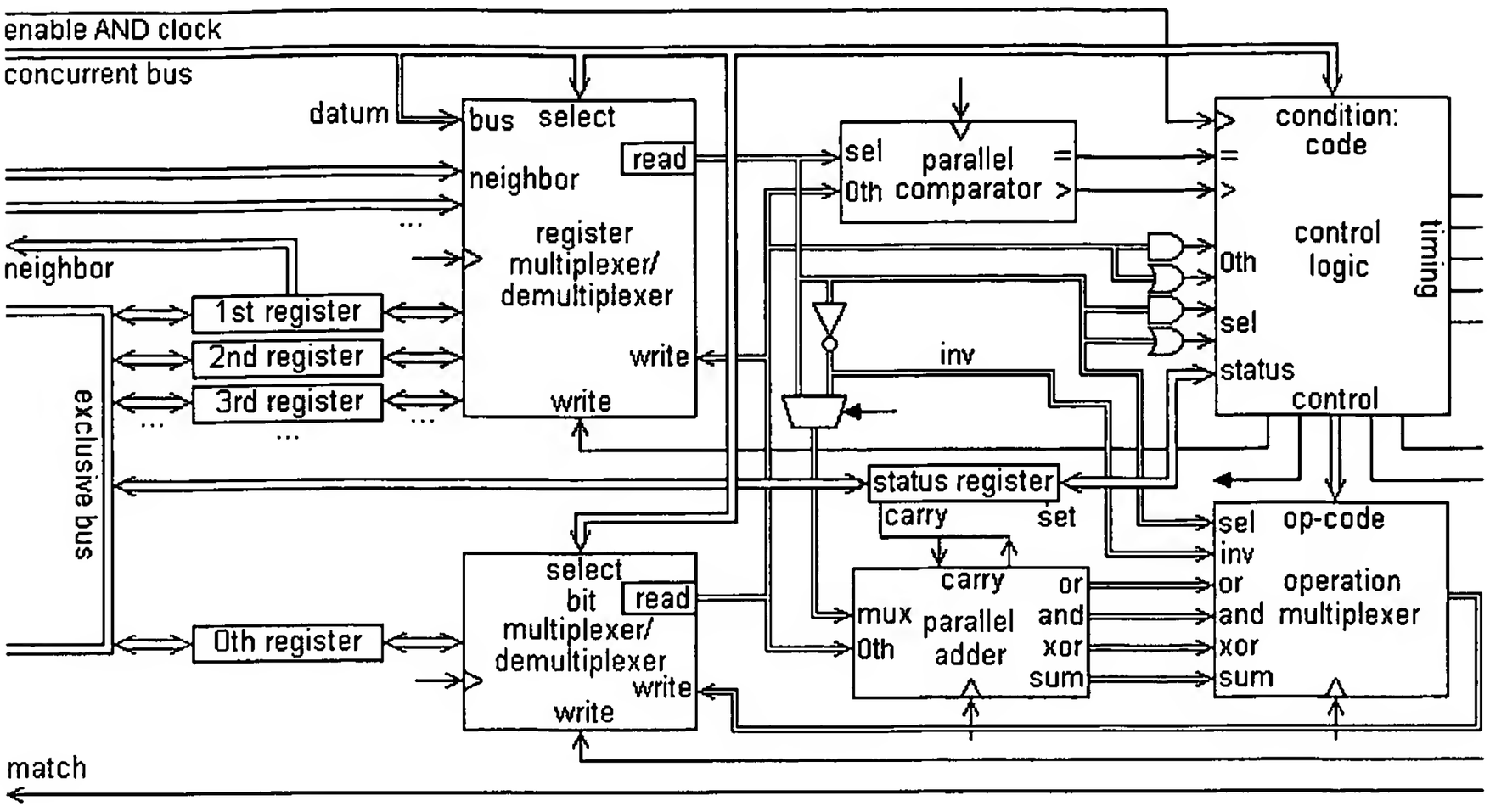

Figure 8: Math Element Construct

The instruction format for the element is "condition: operation width [bit] register[bit]", in which:

- One operand is a bit section of the operation register starting from the first "[bit]" bit, which is selected and cached by the bit multiplexer.
- The other operand is a bit section of the "register" starting from the second "[bit]" bit, which is selected and cached by the register multiplexer. The "register" could be: any of its data registers, its neighboring register, any of its neighbor's neighboring registers, or the datum on the concurrent bus. This operand can be bit-wise inverted.
- The bit width of the two operands is the "width".
- The "operation" is the operation on the two operands. It controls the operation multiplexer, which selects the results of concurrent execution of all operations of the two operands, such as bit-wise AND, bit-wise OR, bit-wise XOR, and value SUM of the two operands.
- The "condition" is the condition for the "operation width [bit] register[bit]" to be executed. It controls the control logic unit, which inputs from a comparator comparing "[bit]" and

"register[bit]", the status register, and the AND or OR combination of all the bits of either "[bit]" or "register[bit]".

The "condition" can be: (1) none, (2) any one of, (3) the AND or OR combination of any one from any two categories of:

- "ANY [bit]", "ALL [bit]": If any bit or all bits of the bit section "[bit]" are positively asserted.
- "ANY register[bit]", "ALL register[bit]": If any bit or all bits of the bit section "register[bit]" are positively asserted.
- <, <=, ==, !=, >=, >: If the corresponding value relation between the "[bit]" bit section and the "register[bit]" bit section is satisfied.
- R, S: If the status bit before this current instruction is negatively or positively asserted.
- E, C: If the carry bit before this current instruction is negatively or positively asserted.

The element of a database memory has no data processing capability. Its instruction set contains at least the following operations:

- ID: Assert positively the match bit output of the element.
- RD: Copy the "[bit]" bit section from the "register [bit]" bit section.
- WR: Copy the "[bit]" bit section to the "register[bit]" bit section when the "register" is either any of the data register or the neighboring register of the element.
- CS, SS: Clear or set the status bit.

The element instruction set for a math memory contains the following additional operations:

- NG: Copy the bitwise inverted "register[bit]" bit section to the "[bit]" bit section.
- ND, OR, XR: Mask the "[bit]" bit section with the "register [bit]" bit section using AND, OR, or XOR bitwise logic.
- AD, SB: Add to or subtract from the "[bit]" bit section, the "register [bit]" bit section and the carry bit, and set the carry bit accordingly afterward.

The instruction format is designed so that multiple instructions can be carried out concurrently if they have no confliction. For an example, the concurrent execution of RD and WR results in exchanging the two bit sections.

The shift operations are achieved when the values of "[bit]" are different for the two bit sections. Multiplication and division may be achieved by a series of addition, subtraction and shift operations. General math operations are also possible.

All element instruction has same length and uses one clock cycle on the concurrent bus, so that the element circuit can be treated as combinational logic.

For simplicity of description of the following concurrent algorithms, the operation registers of all the activated elements are collectively referred to as the operation layer; the neighboring registers of all the activated elements are collectively referred to as the neighboring layer; the status bit of the status registers of all the activated elements are collectively referred to as the status layer. Otherwise, individual element and register are mentioned for exclusive operations. The neighboring layer of the element whose element address is one less or one more than the activated element is called the left layer or the right layer, respectively. So forth.

4.2 Move Content

Moving the content of all elements in an address range by one element in either direction only takes the same number of instruction cycles of moving one element, or simply ~1 instruction cycles. Thus, a database memory or a math memory has all the functionality of a content movable memory.

Though a CP memory may have all the functionality of a less complex CP memory, it is more expensive in terms of each element construct, and probably less powerful in terms of element count. Thus, a CP memory should be suitable for its applications, e.g. a math memory should not be used where only a content movable memory is needed.

4.3 Find and Count Matches

Any matching takes ~1 instruction cycles. The status register can be used for implementing complex matching or memorizing the matching result. Other operations can then be performed on all the matched elements without knowing where they actually are. The matched elements can also be identified and counted using Rule 6, such as limiting the address range to exclude the found matches to enumerate all matches in ~ match count instruction cycles. Thus, the

conventional methods for quick matching when using a conventional memory, such as index table, are no longer required.

Using a math memory, by further sending a corresponding weight factor to all the elements after every step of matching, a degree of match can be accumulated to quantify the match. In this way, the database can be searched using fussy logic.

By matching and counting each section limit one-by-one, the histogram of M sections is constructed in ~ M instruction cycles. The sum and the statistical distribution of the array can be estimated or calculated by similar match-and-count algorithm.

Since each comparison of all elements with their neighbors takes only ~1 instruction cycles, the local maximums in M neighborhood can be found in ~M instruction cycles.

4.4 Local Operations

A special 1D vector of odd-number of items is used to describe the content of the operation layer. The center item describes the content originated from neighboring layer of the element itself and is indexed as 0. The item left to the center item describes the content originated from the left layer and is indexed as -1. The item right to the center item describes the content originated from the right layer and is indexed as +1. So forth. For an example: (A) (1) denotes the content of the neighboring layer; (B) (1 0 0) denotes the content of the left layer; (C) (1 1 0) denotes the content of adding the left layer to the neighboring layer. Two successive operations are additive if both of them use the operation layer accumulatively, such as:

```
(1 1 0) = (1) + (1 0 0);
```

Mathematically, a + operation is defined as:

```
C = A + B: C[i] = A[i] + B[i];
```

The + operation satisfies:

```
A + B = B + A;

(A + B) + C = A + (B + C);
```

When operation layer is copied to or exchanged with neighboring layer, the successive operations are no longer addictive, such as a 3-point (1 2 1) Gaussian averaging algorithm:

1. Copy neighboring layer to operation layer.
2. Add left layer to operation layer.

3. Copy operation layer to neighboring layer.

4. Add right layer to operation layer. The result is in operation layer.

In the above algorithm, without Step 3, Step 4 is also additive to Step 1 and 2, and the algorithm result is (1 1 1). When the result of a first operation A undergoes a second operation B, the overall operation C is expressed mathematically as:

$$C = A \# B: C[i] = \sum_j (A[i+j]\ B[i-j]);$$

The # operation satisfies:

$$A \# B = B \# A;$$

$$(A \# B) \# C = A \# (B \# C);$$

The # and + operations satisfy:

$$(A + B) \# C = (A \# B) + (A \# C);$$

Thus, the above 3-point (1 2 1) Gaussian averaging algorithm is expressed as:

$$(1\ 2\ 1) = (1\ 1\ 0) \# (0\ 1\ 1);$$

And a 5-point Gaussian averaging requiring 6 instruction cycles is expressed as:

$$(1\ 2\ 4\ 2\ 1) = (1\ 1\ 1) \# (1\ 1\ 1) + (1);$$

This concept is extendable to 2D local operations, such as a 9-point Gaussian averaging which requires 8 instruction cycles:

$$\begin{Bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{Bmatrix} = \{1\ \ 1\ \ 0\} \# \{0\ \ 1\ \ 1\} \# \begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix} \# \begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix};$$

Generally, a local operation involving M neighbors takes ~ M instruction cycles.

4.5 Sum

To sum a one-dimensional array of N items, the array is divided into sections, each of which contains M consecutive items. In step 1, all sections are summed concurrently from left to right, in ~ M instruction cycles. In step 2, the section sums, which are at the right-most items of every sections, are summed together serially in ~ N / M instruction cycles. This algorithm can be displayed by the algorithm flow diagram in Figure 9, in which a serial operation is represents by a data flow arrow, and concurrent parallel operations are represents by a data flow arrow with two parallel bars on each side. Each data flow arrow shows the data range of the operation, such as on a section of M items within the whole array of N items. Each series of arrows is marked by a

step sequence number followed by ":", an instruction cycle count preceded by "~", and an operation, such as "1: ~M sum". The instruction cycle counts from consecutive and independent steps are additive, so that the total instruction cycle count is ~ (M + N /M), which has a minimum of ~ √N when M ~ √N.

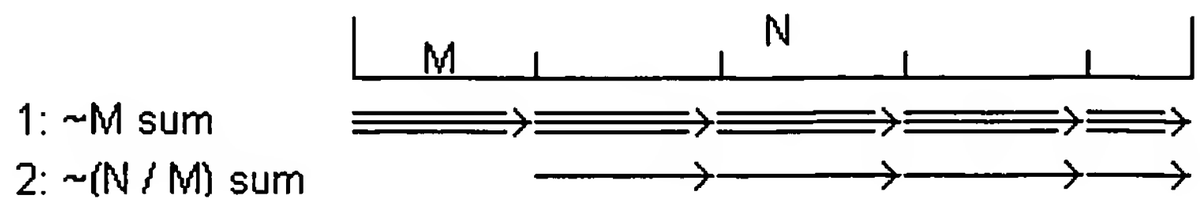

Figure 9: Algorithm Flow Diagram for 1-D Sum

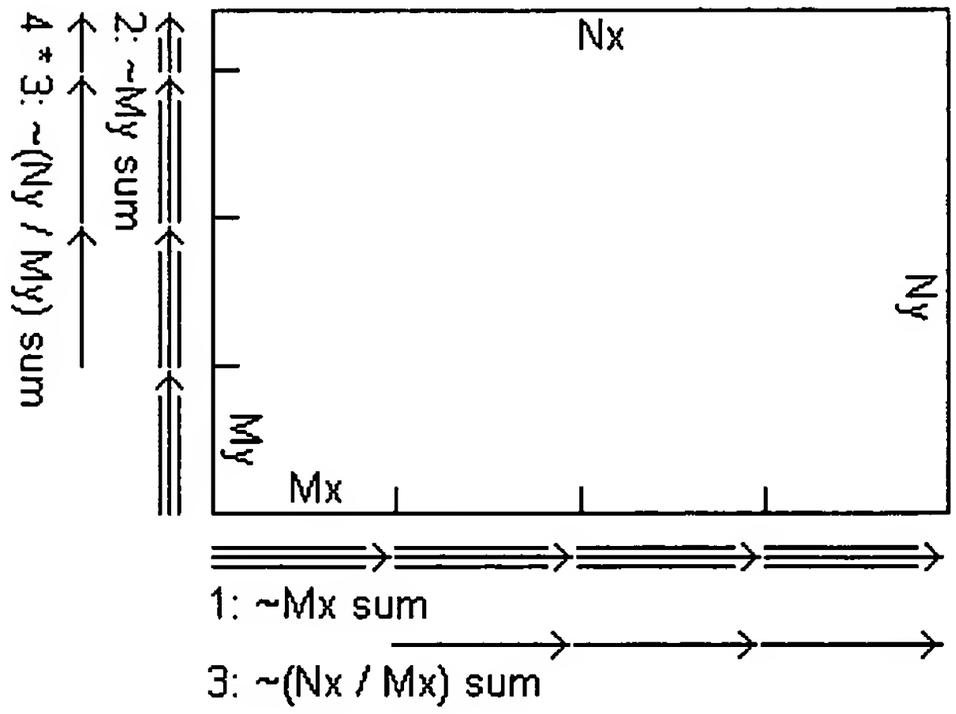

Figure 10: Algorithm Flow Diagram for 2-D Sum

To sum a two-dimensional array of Nx by Ny items, the array is divided into sections, each of which contains Mx by My consecutive items. In step 1, all rows of all sections are summed concurrently from left to right, in ~ Mx instruction cycles. In step 2, all the right-most columns of all sections, each item of which contains a row sums for the section, are summed concurrently from bottom to top. Then the top-right-most items of all sections, each of which contains the section sum, are scanned and summed together serially, with the row and the column direction being the fast and the slow scan direction, respectively. Figure 10 is the corresponding algorithm flow diagram, in which step sequence number "4 * 3" means that a complete step 3 is carried out before each instruction cycle of step 4. The total instruction cycle count for such combination of steps is the product of the individual instruction cycle count of each step. The total instruction

cycle count for the 2-D sum is (Mx + My + Nx /Mx Ny /My), which has a minimum of ~ $\sqrt[3]{}$(Nx Ny) when Mx ~ My ~ $\sqrt[3]{}$(Nx Ny).

4.6 Template Matching

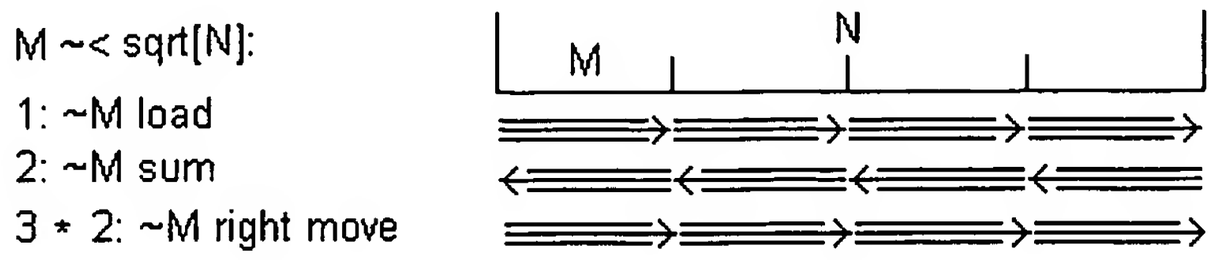

Figure 11: Algorithm Flow Diagram for 1-D Template Matching

To match a template of size M, the array is divided into N / M sections, each of which contains M consecutive items. The algorithm diagram is shown in Figure 11. In Step 1, the template to be matched is concurrently loaded to all sections in ~M instruction cycles. Then the point-to-point absolute difference is calculated concurrently for all points in ~ 1 instruction cycles, which is omitted from the algorithm flow diagram. In Step 2, the differences in all sections are summed concurrently from right to left in ~ M instruction cycles, to obtain the difference values of the array to the template at the left first positions of all sections. In the first instruction cycle of Step 3, the templates in all sections are shifted right concurrently by one item, to calculate the difference at the left second positions of all sections, and so forth. Thus the total instruction cycle count is ~ (M + M^2) ~ M^2.

Similar algorithm can be carried out in a 2-D array of size Nx by Ny stored in a 2-D math memory for a 2-D template of size Mx by My. The algorithm diagram is shown in Figure 12. In step 2 *1, the template to be matched is loaded to all sections concurrently. The first instruction cycle of Step 3 sums the point-to-point absolute difference of each row of each section at the left-most column of the section. The first instruction cycle of Step 4 moves the template right by one column. The first complete application of Step 4 * 3 fills all the columns with the sums of row difference of the corresponding sections. The first instruction cycle of Step 5 results in the matching of the template to the bottom-most row of each section. The first instruction cycle of Step 6 moves the template up by one row. The Step 4 * 3 is carried out again except that the Step 4 is carried out from right to left this time. The total instruction cycle count is ~ (Mx My + (Mx^2 + My) My) ~ (Mx^2 My).

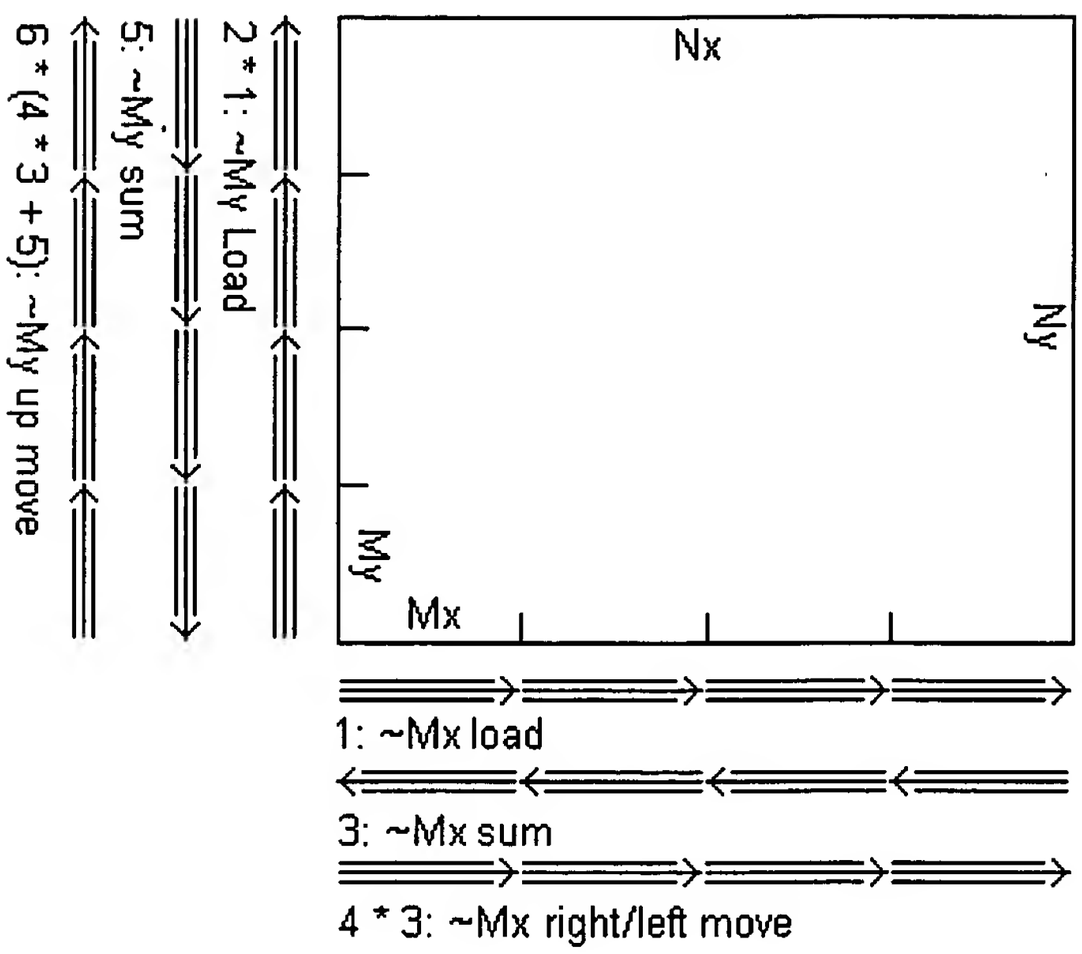

Figure 12: Algorithm Flow Diagram for 2-D Template Matching

The instruction cycle count is reduced from ~ (N M) to ~ M^2 for 1-D template matching, and it is reduced from ~ (Nx Ny Mx My) to ~ (Mx^2 My) for 2-D template matching, thus no longer depends on the original image size. It may be small enough now for the template matching algorithm to be carried out in real-time for a lot of applications, such as image databases.

4.7 Sort an Array

By asking all elements to identify themselves if their left layer is larger than their neighboring layer, the disorder items, which are the items stored in the neighboring layer that need to be sorted to small-to-large order, can be all found immediately, and thus a sorting algorithm can stop immediately if no further sorting is required. The disorder item count also guides the sorting direction. If initially the disorder item count for small-to-large sorting is more than that of the large-to-small sorting, the array should be sorted into large-to-small order—to sort an array in either order is functionally equivalent. Thus, the worst case for sorting, to sort a nearly sorted array into the other order, can be avoided.

Only ~1 instruction cycles are required to exchange concurrently all even and odd numbered neighboring items once toward small-to-large order. Alternatively repeating exchanging all (1) even and odd and (2) odd and even numbered neighboring items makes a local exchange sorting

algorithm, which is good at removing random local disorders. Using local exchange sorting algorithm, any array can be sorted in no more than ~N instruction cycles.

Contrary to the local exchange sorting algorithm, a global moving sorting algorithm removes disordered items in a nearly sorted array and inserts them to proper place, by analyzing the "topography" of the sorting disorders. Figure 13 describes point defects in the topography:

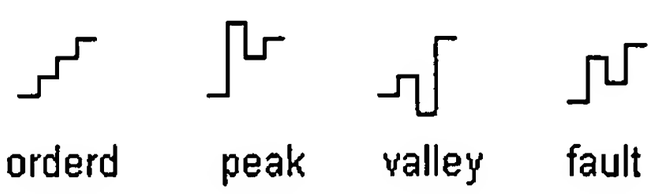

Figure 13: Point Defections in a Otherwise Sorted Neighborhood

- Peak: It is an insertion of a larger item into an otherwise ordered neighborhood. To restore order, the peak item can be moved to the left of the left-most item to its right which is larger than it, or to the right end of the sequence in ~2 instruction cycles.
- Valley: It is an insertion of a smaller item into an otherwise ordered neighborhood. To restore order, the valley item can be moved to the right of the right-most item to its left which is smaller than it, or to the left end of the sequence in ~2 instruction cycles.
- Fault: It is an exchange of two neighboring items. To restore order, the two faulty items can be exchanged in ~1 instruction cycles.

The concurrent detection all of the above point defects in each 4-item neighborhood requires ~4 instruction cycles. The simplest global moving sorting algorithm only detects and removes point defects, leaving more complicated defects to the local exchange sorting algorithm. The ratio of the disorder item count to the total item count is defined as disorder ratio, which can be used to choose between these two sorting algorithms. Using the instruction set defined in section 4.1 Element Construct", Figure 14 left shows the simulated experimental instruction cycle count of either algorithm in sorting a unique sequence of various sizes with introduced random disorders of various initial disorder ratios. It shows that for a given total item count, when the initial disorder ratio is less than a threshold disorder ratio, e.g., 0.02 for 2^{14} items, the simplest global moving sorting algorithm is more effective. The reason is that the instruction cycle count of the local exchange sorting algorithm is determined by the largest distance of all disorder items from their respective proper places (which is close to N when the disorder count is more than a few), while

when the point defects are the majority defects at low initial disorder ratios, the instruction cycle count of the global moving sorting algorithm is approximately proportional linearly to the initial disorder count. The threshold disorder ratios seem to obey a power relation in regard to the total item count, as shown in Figure 14 right, so that it can be calculated for a sequence of any size. When the initial disorder ratio is higher, the local exchange sorting algorithm can be used first to reduce the disorder ratio below the threshold, the global moving sorting algorithm can be used secondly to accelerate sorting.

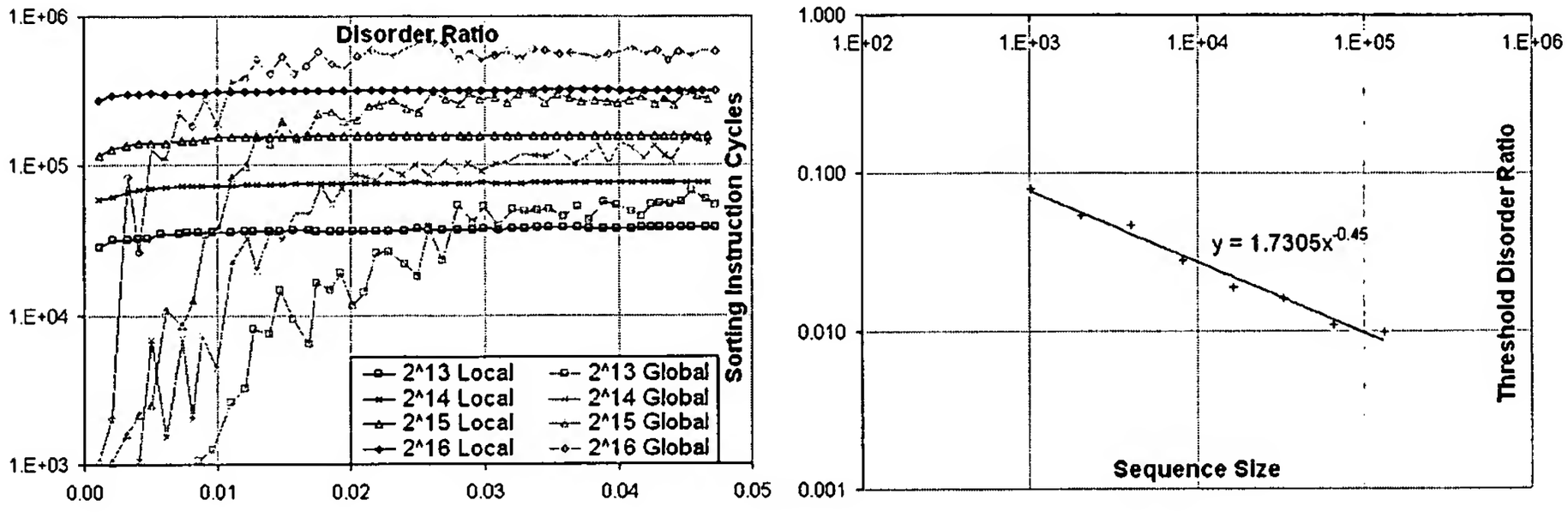

Figure 14: (left) Comparison of instruction cycle count using either local exchange sorting or global moving sorting for different disorder ratio. (Right) the threshold disorder ratios for different total item count.

4.8 Thresholding

With its multiple dimensions of data, image processing and modeling generally requires large amount of calculation, which is proportional to the size of data in each dimension [11].

Using a conventional bus-sharing computer, the instruction cycle count is linearly proportional to the amount of calculation [11]. Thus, to solve a problem in a realistic time period, thresholding is frequently used to ignore large amount data for the subsequent processing. Thresholding is a major problem [1], because proper thresholding is difficult to achieve, and thresholding in different processing stages may interact with each other.

Using a math memory, the instruction cycle count is decoupled from the amount of calculation, and is independent of the size of data in each dimension. Thus, thresholding can be

used only in last stage to qualify the result. Also, thresholding itself has been reduced to ~1 instruction cycle operation.

4.8 Line Detection

Due to neighbor-to-neighbor connectivity, math 2D memory can treat line detection problem as a neighbor counting problem. To detect edges line of pixel length L lying exactly along X direction left to each pixel, the neighbor count algorithm is direct:

1. All pixels concurrently subtract the raw intensity of their bottom layer from that of their top layer, and store the result in the neighboring layer.
2. All pixels concurrently sum the neighboring layers of their L left neighbors together with their own. The absolute value of the result indicates the possibility of an edge line starting from that pixel, while the sign of the result indicates whether the edge is rising or falling along the Y direction.

To detect edge lines with a slope of (My / Mx), in which Mx and My are two integers, each pixel defines a rectangular area of Mx by My pixels denoted as (Mx * My), and the line which connects the pixel and the furthest corner of the area has the slope of (My / Mx). Similar to obtaining the section sums in a sum algorithm, a messenger starts from furthest corner of the area, walks (Mx + My) steps along the line until it reaches the original pixel. In each of its stop, in a predefined fashion, if the pixel is on the left side of the line, its intensity is added to the messenger; otherwise, its intensity is subtracted from the messenger. When reaching the original pixel, the value of the messenger indicated the possibility and the slope direction of the edge line segment which connects the original pixel and the furthest corner of the (Mx * My) area. Thus, it is called the line segment value of the pixel for the (Mx * My) area. This accumulating process is carried out concurrently for all the pixels of the image, independent of image sizes. Figure 15 shows the (4 * 3) area to detection a line with a slope of (3/4) passing the original pixel at 0. The accumulation processing is from pixel 7 to pixel 0 in sequence, with the raw intensity of pixel 1, 3, and 5 to be added to, and those of pixel 2, 4, and 6 to be subtracted from the messenger.

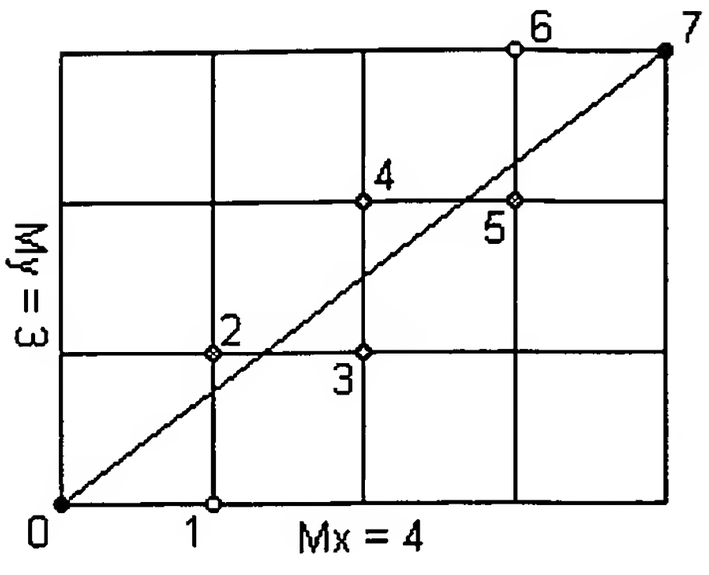

Figure 15: 2-D Line Segment Detection Using Messenger

Given an angular resolution requirement, a {(Mx, My)} set can be constructed to detect all line segments on an image, each element of which can be determined by a corresponding line segment detection algorithm. Figure 16 left shows a set of lines whose pixel spans are exactly 7 in walking distance. It also shows the walking distance envelope of 7. For such a line set of walking distance D, the angular resolution is ~(2/D) along the 45-degree diagonal direction; the total instruction cycle count is ~ D^2, independent of the image size.

To reduce the instruction cycles for detecting the set of lines of the given angular resolution, starting from a {(Mx, My)} set of D in walking distance, a circle of radius ~(D/√2) in real distance may be used to guide the starting walking pixels for the messengers. Figure 16 right shows such a set of lines whose pixel spans are ~5 in real distance. It also shows the real distance envelope of 5.

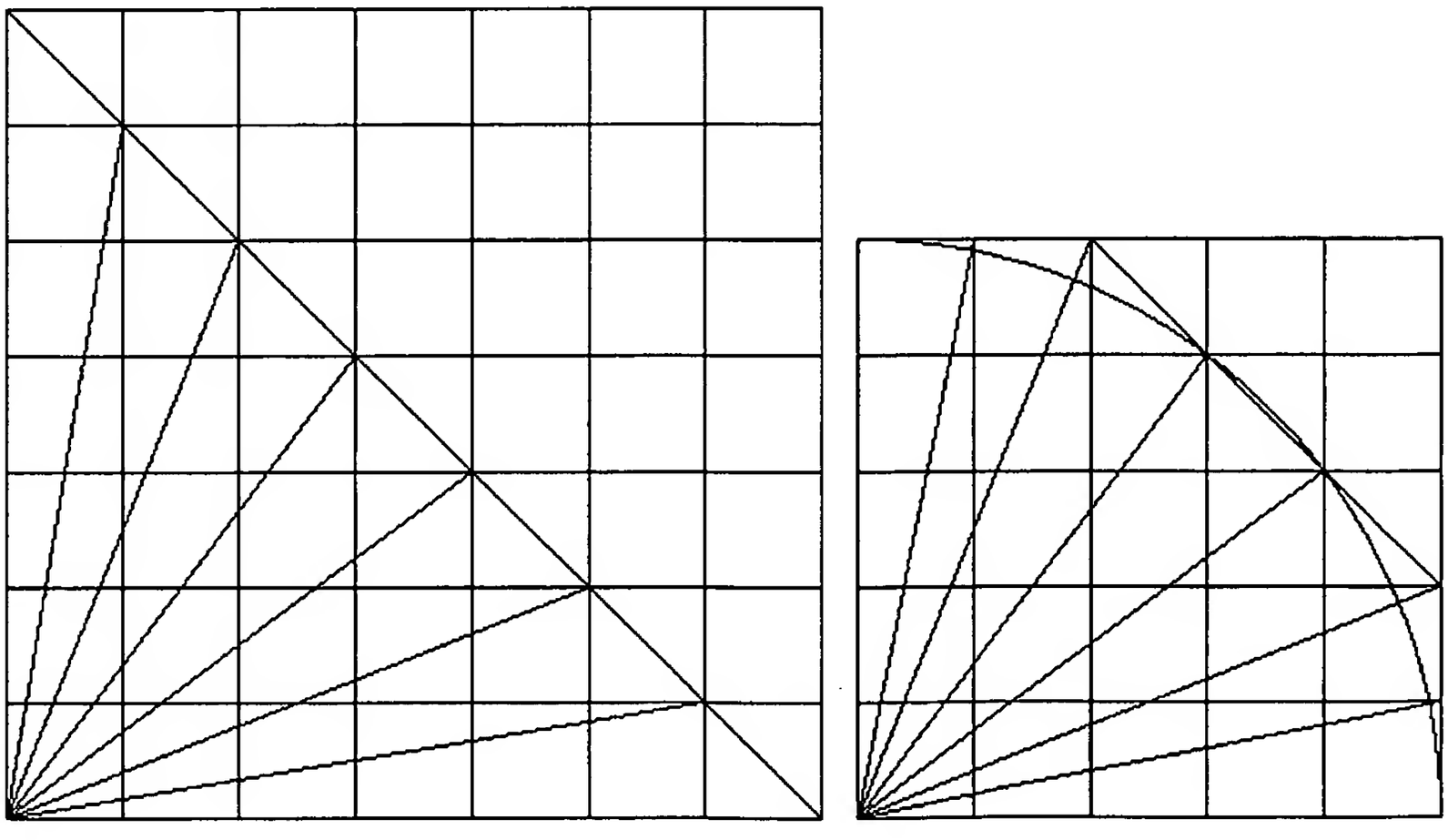

Figure 16: (Left) A Set of Lines of Pixel Spans 7 In Walk Distance. (Right) A Set of Lines of Pixel Spans 5 In Real Distance.

As a result of line detection, each pixel is marked by the best line segment value together with its corresponding (Mx, My) area. More advanced graphic analysis can then be carried out based on these line segment values [1].

5 Discussion

The following table summarizes the performance improvement of concurrent processing memory in term of required instruction cycles over traditional bus-sharing architectures [3] and traditional SIMD PE architectures [11][12][13] (with only the best performance for each algorithm among all different SIMD construct listed):

Array of size N	Bus-sharing	Traditional SIMD PE	CPM
Find a Match	~N	~1	~1
Compare a Value	~N	~1	~1
Enumerate M Values	~N	~M	~M
Insert and Delete	~N	~√N	~1
Construct a histogram of M sections	~N	~M	~M
Match a Template of size M	~N * M	~M	~M
Filter with a filter of size M	~N * M	~M	~M
Find Local Limit within each M neighbors	~N * M	~M	~M
Find Global Limit	~N	~N	~√N
Sum	~N	~N	~√N
Sort	~N * Log(N)		~N

Better programmability is the major advantages of the SIMD parallelism [10][11][12][13] over the prevailing MIMD parallelism [4][5][6][7][8][9]. As a member of SIMD parallelism, by introducing coherent collective element activation rules, the CP memory has achieved more general applicability. Except silicon integration, which is currently undergoing rapid development,

all technology required are off-the-shelf. The proposed end product is completely compatible with the current common CPU/memory architectures. The recent development of content addressable memories [14] shows great powers and promises in searching millions of bits concurrently, demonstrating the feasibility of unifying data storing and processing on a large scale. The design of CP memory may find its use in future as an alternative to higher clock rate in speeding up execution, though its usability and applicability has to be tested by experimentations.

Acknowledgement

This work was finished during the nine month when Dr. Chengpu Wang was laid off and could not find a job. It is possible only because of the unquestioned supports and unconditional loves from Dr. Chengpu Wang's wife, Dr. Yingxia Wang. Dr. Chengpu Wang also feels indebted to the encouragements and valuable discussions with Dr. Nick Tredennick from Gilder Technology Report, Prof. Pao-Kuang Kuo from Wayne State University, Prof. Sangjin Hong and Prof. Tzi-cker Chiueh from SUNY at Stony Brook, and the organizers of PDPTA 2003, Prof. Hamid R. Arabnia from University of Georgia in particular. Finally, the authors feel obliged to the editors and reviewers of this journal for their acceptance of this paper from this unusual source and their works to finalize this paper.

Appendix 1. Content Matchable Memory

A content matchable memory concurrently matches the data stored in each of its element with a common value to be matched in ~1 instruction cycles. Thus, the conventional methods for quick matching when using a conventional memory, such as index table, are no longer required. The major improvement of content matchable memories over traditional content addressable memories [16] is to activate and inactivate multiple elements instantly according to Rule 4, instead of individually, to greatly increase the task switching speed.

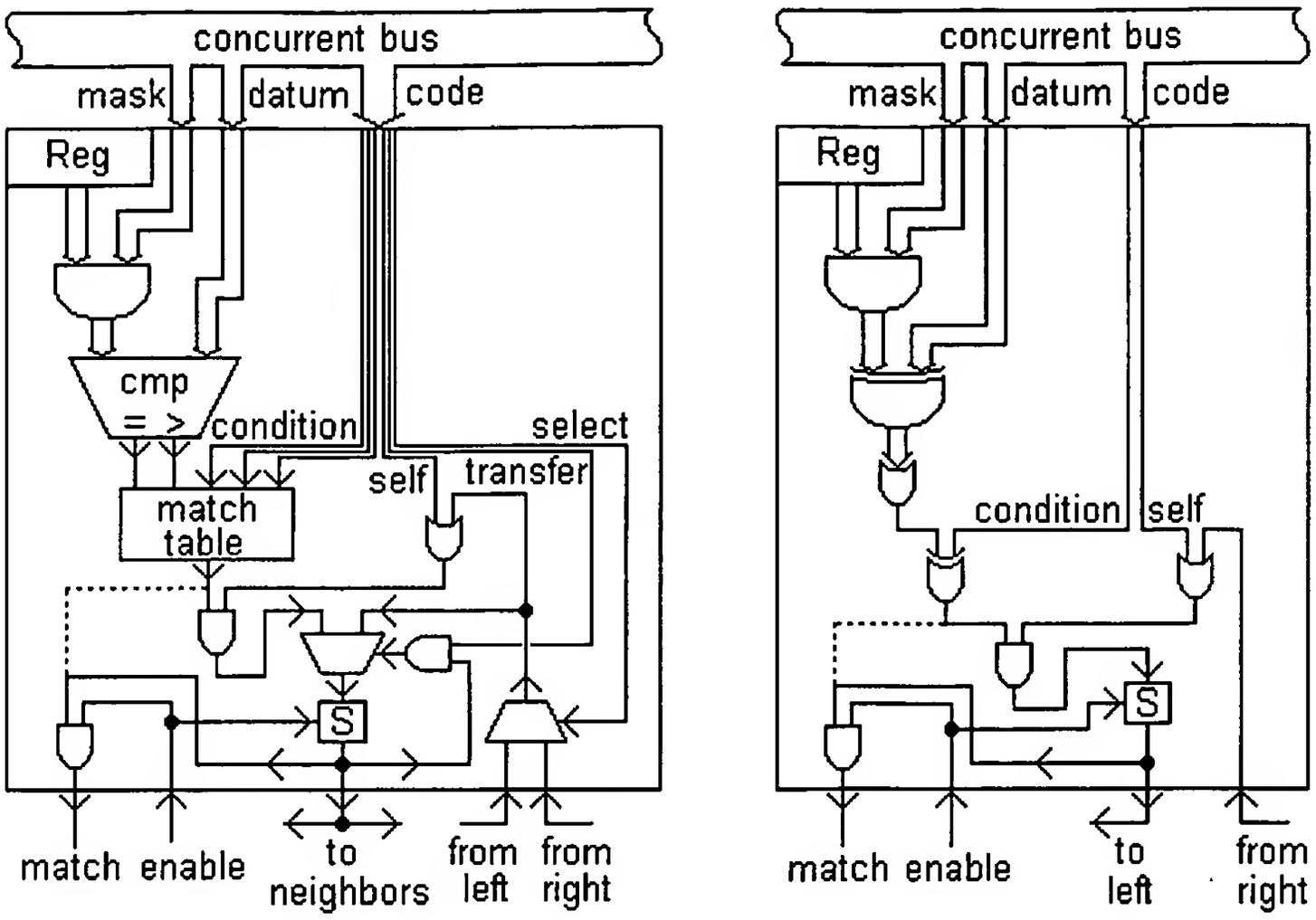

Figure 17: Structures of Content Matchable Elements.

A1.1 Content Comparable Memory

Figure 17 left shows a content comparable memory element. During match, the concurrent bus sends to every element: (1) a mask, which masks the content of the only addressable register of the element; (2) a datum, whose value is compared with the masked data at a comparator; and (3) a condition code of comparison requirement, which is matched against the output of the comparator at a match table. The bit output from the match table is AND combined with the enable bit input to assert the match bit output of the element.

Additional logic allows the value matching across elements when each of array items to be matched is saved by several neighboring elements in the order of significance. When the element is enabled, the bit output value of the match table is saved into a one-bit S register, which is connected to neighboring elements through neighborhood connection. The concurrent bus sends three more instruction bits. The instruction bit "select" selects the S register of either the left or the right neighboring element to the output of a multiplexer. Through an OR gate, when the instruction bit "self" is positively asserted, the S register of the element is positively asserted if a matched is found by the match table; otherwise, the selected neighboring S register also has to be positively asserted. Through a multiplexer and a AND gate, when both the

instruction bit "transfer" and the S register of the element itself are positively asserted, the value of the selected neighboring S register is saved into the S register of the element; otherwise, the bit output value of the match table is saved.

For simplicity of discussion, the width of the addressable register is a byte, and the value to be compared is unsigned. The comparing algorithm is the following: (1) The S register of each of all the most significant elements is positively asserted if the element contains the most significant byte to be matched. (2) In the order of decreased significance, the S register of each of all the successive elements is positively asserted if (A) the element contains the corresponding byte to be matched and (B) its neighboring S register of immediately higher significance is positively asserted. In this way, the S register of the least significant element of each of all array items holds the result of an "==" or "!=" comparison. (3) In the order of increased significance, if each of all the S registers is already positively asserted, it copies from the neighboring S register of immediately lower significance; otherwise, it saves the desired matching result of the element with the corresponding byte to be matched. In this way, at the end of the comparison, the S register of the most significant element of each of all array items holds the result of a "<", "<=", ">=", or ">" comparison.

A1.2 Content Searchable Memory

To limit function to searching a value among neighboring elements, such as string search, the element construct can be further simplified. Figure 17 right shows a content searchable element. The concurrent bus only has two bits: (1) a condition code bit of "==", and (2) a "self" instruction bit. The searching algorithm is the following: (1) The S register of each of all the elements is positively asserted only if it equals the right-most byte to be searched. (2) In the right-to-left order, each of all the successive elements has its S register positively asserted only if (A) the element contains the corresponding byte to be searched, and (B) its right neighboring S register is originally positively asserted. In this way, a positively asserted S register at the end of the search indicates that the element is the left-most element of the neighboring elements which together hold a value to be searched. The carry number of the content searchable memory is a constant of 1 for Rule 4.

Although a content searchable memory has identical functionality as a content addressable memory [14], it has the following advantages: (1) the size of objects to be matched is no longer limited to the bit width of memory elements; (2) If each memory element can have other registers that can be copied to the searchable/comparable register, different tasks can use different register set, thus allow multitasking and instant task switching.

Reference

[1] E. R. Davies, Machine Vision: Theory, Algorithms, Practicalities (Academic Press, 1990)

[2] T. J. Fountain, Parallel Computing: Principle and Practice (Cambridge, 1994)

[3] John P Hayes, Computer Architecture (McGraw-Hill, 1988)

[4] John L. Hennessy, David A. Patterson, Computer Organization & Design (Morgan Kaufmann 1998)

[5] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In: Supercomputing, November 1999.

[6] Basilio B. Fraguela Jose Renauy Paul Feautrierz David Paduay Josep Torrellas. Programming the FlexRAM Parallel Intelligent Memory System, PPoPP 2003.

[7] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In: ISCA, June 2000.

[8] The Berkeley Intelligent RAM (IRAM) Project, Univ. of California, Berkeley, at http:// iram.cs.berkeley.edu.

[9] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. IEEE Communications Magazine, Vol. 35, No. 1, pp80-86. January 1997.

[10] Mark Oskin, Lucian-Vlad Lita, Frederic T. Chong, Justin Hensley and Diana Keen. Algorithmic Complexity with Page-Based Intelligent Memory. Parallel Processing Letters Vol 10. No 1 (2000) pages 99-109.

[11] R. J. Offen, VISL Image Processing (McGraw-Hill, 1986)

[12] Caxton Foster, Content Addressable Parallel Processors (Van Nostrand Reinhold, 1976)

[13] A. Krikelis (Editor), C. C. Weems (Editor), Associative Processing and Processors (IEEE Computer Society Press, 1997)

[14] L. Chivin & R. Duckworth, Content-addressable and associative memory: Alternatives to the ubiquitous RAM. IEEE Computer Magazine, p51-64, July 1989

[15] C. P. Wang, and Z. Wang, A Smart Memory Design. In: Parallel and Distributive Processing, Technology, and Application, June, 2003.

[16] Herbert L. Dershem, Michael J. Jipping, Programming Languages: Structures and Models (Wadsworth Publishing, 1990).